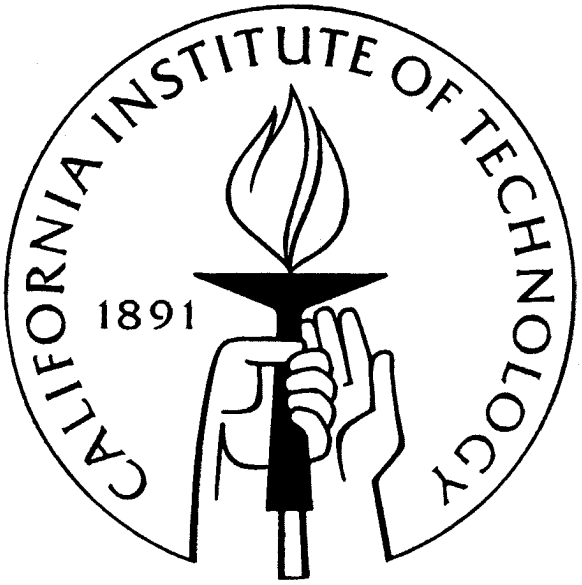


A Compiler for a Subset of Modula-3

Donald Pinkstone III

Computer Science Department
California Institute of Technology

Caltech-CS-TR-94-17

A Compiler for a Subset of Modula-3

Donald Pinkston III

June 8, 1994

1 Introduction

It is the purpose of this paper to detail the current status of the Modula-S compiler, as well as the backgrounds and solutions of the major areas I encountered. The purpose of this project was to create a compiler for a subset of the Modula-3 programming language, while exploring various ways to effectively and flexibly implement the chosen constructs and statements of the Modula-3 programming language. I began with a very simple Pascal-S compiler contained in Jan van de Snepscheut's *What Computing Is All About*, and after converting the code to Modula-3, I began the process of modifying the compiler to support a subset of the Modula-3 programming language. Because the scope of the original compiler was very limited, I was able to research and implement many of the basic Modula-3 statements. I encountered several interesting areas during my progress, including the implementation of runtime error-checking and open array structures. These and other topics are discussed in some depth in the remainder of this paper.

2 Definition of Modula-S, a Subset of Modula-3

The Modula-S programming language includes a relatively basic yet useful subset of the Modula-3 programming language. The language has slowly expanded over time, as the intent was to create a language as similar to Modula-3 as possible with various restrictions.

Although Modula-S is intended to be a subset of Modula-3, the language also has several characteristics of the Pascal-S language [vdS93], for reasons detailed later.

2.1 Identifiers and keywords

All identifiers in Modula-S are case-sensitive, as in Modula-3. All reserved keywords are capitalized.

2.2 Basic data types

The basic data types `INTEGER`, `CHAR`, and `BOOLEAN`, as well as their associated functions and operations, are all completely supported. Enumerations and subrange types are also fully supported, except for the inclusion of set types. Unlike Modula-3, the real number types and their associated functions are not supported. Also, no pointer or reference types are available to the user, as well as variables of type `PROCEDURE`.

2.3 Data structures

The Modula-S language supports the two basic data structures of arrays and records, also to varying degrees. Modula-S allows the use of multiple- dimensioned, fixed size arrays and variable-length open arrays; however, open arrays are only allowed to have one open (variable) dimension and are only allowed to be used as `VAR`-mode procedure arguments. All normal array and record operations are allowed in the Modula-S language; however, this specific implementation does not support the use of open arrays with the `WITH` statement. Modula-S does not support sets, `TEXT` strings, or object types.

2.4 Program structure and I/O

Programs may consist of only one module and cannot import from or export to any other module or interface. Console input and output are handled through the two Pascal-like commands `READ` and `WRITE`. `READ` allows programs to accept `INTEGER` and `CHAR` input but not text strings. Also, `WRITE` allows programs to print `INTEGER` and `CHAR` types, as well as all related subranges. All Modula-3 escape codes are supported with the `WRITE` statement. There is no support for file or other I/O operations.

2.5 Declarations

As in Modula-3, programs are allowed to include any number of constant, type, variable, and procedure declarations; however these declarations must be made in the particular order of constants, types, variables, procedures. Any other order will result in a compilation error. Procedures containing their own declarations must also maintain this order. All variable declarations may have initial values specified, as in Modula-3.

2.6 Exceptions

Modula-S currently includes no support for error-trapping or exceptions, even though runtime errors will be raised by programs encountering various situations. However, because of the lack of interfaces and the extreme limitation of I/O, the lack of exceptions is not very restrictive.

2.7 Procedures

Procedures in Modula-S also vary from Modula-3 in several ways. Declarations are structured identically to Modula-3; however because exceptions are not supported, the `RAISES` section is excluded. Argument variables can be addressed by the same three modes in Modula-3. There is one additional restriction on the modes that can be used besides those present in Modula-3: open arrays must be passed using `VAR`-mode addressing (see open array section for reasons).

Argument variables have several other restrictions on them as well. There is currently no support for default values. Also, procedure arguments must be placed in the proper positional order; the parameter names cannot be included. Other than these restrictions the procedure declarations and uses are identical to Modula-3.

Procedures are allowed to be recursive; however because the compiler is a single-pass compiler, procedures cannot be called until they are declared in the code. The Pascal command `FORWARD` is not supported for these situations.

2.8 Commands in general

Other than the `WITH` statement, most other Modula-3 statements are fully supported by Modula-S. The only non-Modula-3 commands are `READ` and `WRITE`, which help facilitate I/O.

3 The Modula-S Compiler

In producing a compiler for the Modula-S programming language, several interesting problems were encountered. Some involved the structure and methods of the original Pascal-S compiler, and others centered around the proper implementation of the Modula-S statements themselves. Several of the larger and more interesting topics, their backgrounds, and their solutions are detailed in the following sections.

3.1 Single-Pass Compiler

The Modula-S compiler is different from most Modula-3 compilers in that it compiles the program after only a single pass over the code. Although this leads to several limitations in the structure of programs, it is interesting to note that a useful, although small, number of Modula-3 instructions *can* be created using only a single pass of a compiler. However, there are many drawbacks of using only a single pass compiler with Modula-S, obstacles which can become quite limiting in certain situations.

3.1.1 Limited program structure

The most obvious problem arising from using a single-pass compiler is that procedures and other constructs must be declared before they can be used in the code. Simply put, the compiler must be able to determine what it is dealing with before it encounters any instances of it in the code. This is unlike a multi-pass compiler which can determine all necessary information on the first pass and then go back later. Although this does not often create a huge dilemma in the use of the language, it can still lead to problems in certain situations. Procedures that are mutually-recursive (e.g. procedure *a* calls procedure *b* calls *a*) cannot be used because one of the procedures will contain a call to a procedure that is not yet defined. This can become a great hindrance in some situations, and is an inherent weakness of this Modula-S compiler. Also, variable and type declarations must also be declared before they can be used, unlike most Modula-3 implementations. However, this does not present as large of a problem, and is usually easily worked around.

3.1.2 References and recursiveness

A single-pass compiler can still be used with many recursive structures, as long as those structures are self-recursive. The main reason for this is that often the size of such structures can be determined easily by the compiler. Recursive procedures are easily supported because the size and type of their return value, if any, is stated in the procedure declaration. Recursive references such as linked lists are quite simple to implement as well, for the compiler already knows the type and size of the pointer in the recursive definition. The most important point is that the `TYPE t =` has already been parsed, and the compiler has already created a type entry and ascertained the important information for the recursive reference when it is encountered again, and because the general information is already known, the recursive reference presents no problem whatsoever.

Perhaps another interesting area of study would be to examine creating a "looser" Modula-3 compiler; one that is less restrictive if all identifiers are not previously defined. By operating on certain assumptions, the size and type of procedure return values or recursive definitions could be guessed at from their context, leading to a more complete inclusion of normal Modula-3 structures. This could lead to other problems, however, such as errors which are not recognized at the proper location or not at all. A compiler that implements such strategies to gain this small amount of added functionality would be more useful if simply converted to a multi-pass compiler.

3.1.3 Code optimization

The Modula-S compiler has other limitations as well. The generated code is optimized to a degree by a peephole optimizer which is also single-pass. Creating

a procedure to make multiple passes over the generated code would result in a more complete optimization.

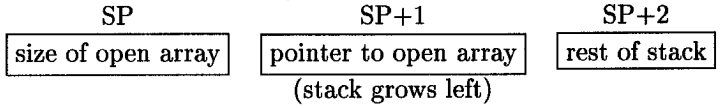
3.2 Open Arrays

Although variable-size structures are not implemented in any greatly useful way in Modula-S, they are included to a certain extent with the limited use of open arrays. Open arrays are greatly simplified in Modula-S: they are only created and used in two main situations, and they are only variable-length in one dimension. The full spectrum of open array use is not included in Modula-S because references have not been included. Also, open arrays are only open in one dimension because the concepts used in implementing one-dimensional open arrays are very similar to those for making multiply-dimensioned open arrays, and the purpose of the project was to explore the new constructs, not to fully implement the Modula-3 language.

3.2.1 The open array descriptor

The descriptor which an open array would use is really quite simple, mainly because open arrays are limited to being variable-length in only one dimension. The descriptor on the stack would then point to the actual array (somewhere in memory), as well as describing its size. This structure consists of two words, the size of the open array in elements (lower word), and a pointer to the start of the array (higher word). This is perhaps not the best possible order for the descriptor; a better method may be to have the pointer come before the array size on the stack, especially if there were multiple open dimensions. However, with the current uses of open arrays, having the open array size first on the stack is the most effective method. Also, the original idea for open arrays was to have the array size in words, allowing the code to be simpler because it would not have to compute the size-in-words from the size-in-elements. However, because the size-in-elements is used so often, uch as in index-checks and open array type checking, the method of storing the open array's size in elements was chosen.

The form of an open array:



3.2.2 Open arrays and VAR-mode addressing

Open arrays may be used only in VAR-mode parameter declarations, such as the following:

```
PROCEDURE whatever(VAR array:  ARRAY OF sometype) =
```

No other method is allowed, simply to keep from having to greatly modify the structure of the stack machine. If, for example, **VALUE**-mode open arrays were allowed, the array would have to be copied to a heap of some sort so the pointer would no longer point to the original array. The open array cannot be duplicated on the stack because the stack's size would no longer be predictable at compile time, leading to a whole slew of other problems. Therefore, the implementation of open arrays is limited to those times when open arrays will be addressing arrays which are already in existence, such as the above procedure declarations as well as uses of the **SUBARRAY** function.

3.2.3 Implementation of open arrays

Several changes were made to the compiler to include open arrays, some major and others minor. Firstly, the type table entries were changed slightly, including a new flag to indicate whether an array is variable-length in the first dimension or not. Secondly, the procedure which loads the actual address of the variable passed to it was changed so that when an open array variable is encountered, the entire descriptor would be loaded on top of the stack, not just the first word of it.

The procedure *selector* was modified slightly to allow indexing of open arrays. The indexing of open arrays is quite similar to fixed length arrays, except that the first index value must be checked to insure that it is actually inside of the open array. Also, if there is any indexing done on an open array, *selector* automatically removes the array's size (converting it to a normal, fixed-length array pointer) as open arrays are only open in the first dimension. Going down any number of dimensions in the open array will result in a fixed-length array with a known size; therefore the array's size is no longer needed on the stack.

3.2.4 Scope of open array usage

Open arrays can be used basically anywhere fixed-length arrays are used. These areas can be grouped into several major sections: comparisons and assignments, the various internal functions, and procedure declarations and calls. Each of these major areas present different problems to address in the use of open arrays, the solutions of which are usually simple and straightforward.

3.2.5 Comparisons and assignments

Open arrays are rather easily integrated into normal array comparisons and assignments. Normally, arrays and other variables are checked by the compiler for type compatibility in a procedure called *mustbe*. This procedure is passed the two types in question and creates an error if the types are incompatible. However, this method of checking types at compile-time is unusable with open arrays because the array sizes are variable at runtime; the only check that can

be performed at compile-time is the compatibility of the element types. Therefore, new code must be generated by the compiler to check these open-array types before the operations are performed. The action is somewhat simplified by the Modula-S language itself; because open arrays are only variable in one dimension, the needed checks are greatly simplified and the additional code is kept much smaller.

The procedure to check open array types and sizes has a secondary function as well. To allow open arrays to be handled as normal arrays (to limit the amount of new compiler code), this procedure also has the function of converting the open array descriptors to normal array descriptors, therefore allowing them to be handled as normal, fixed-length arrays. The procedure can then be called every time by the fixed-length array-handling sections of the compiler, and after the procedure has created its checking code (if needed), the array-handling sections can simply go on as if everything is a fixed-length array.

The new procedure, called *checkarrays*, is passed the types of the two arrays, with the first type being the first array expression loaded and the second type being the second array expression loaded. This way the compiler code can know in what order they have been loaded, and exactly what must be done to check and convert the arrays, if needed. There are four possibilities when *checkarrays* is called: both arrays are fixed-length; the first one is open but the second is fixed; the first one is fixed and the second one is open; and both are variable-length. If the first possibility occurs, no new code is needed. However, the last three instances require various small additions to the executable code to check the open arrays for compatibility.

There is actually very little to note about these checks, except for the final instance in which both are open arrays. In the second and third instances the open array is basically "converted" over to the same type as the fixed-length array (as long as the element types match), so the procedure returns a fixed-array type in place of the original open array type. However, if both are open arrays then there is no way to determine the size of the open arrays after they are converted to normal array descriptors on the stack, so the size of the arrays in words is left on top of the stack as well. Then when the actual comparison or assignment is performed, the size of the arrays is already present on the stack for the subsequent opcodes, either `eqal` (comparison) or `move` (assignment) to work with. Other than this special case, the open array sizes can be predicted with the fixed-length arrays they are being used with, since they must be compatible for the runtime code to not result in a runtime error.

Another alternative for processing open arrays was also explored. Instead of converting the open array descriptors to fixed-length array descriptors before performing the desired operations, *all* arrays were given open array descriptors. After parsing and loading, every array would have its size in elements loaded as well. The former method was chosen over this one, however, because the added overhead of handling the open array descriptors became much larger than the current array-processing code. The former solution is much more useful with this

situation, since compiler code and the required stack space are all kept much smaller. Also, the generated programs are much smaller and faster running, since only the necessary runtime array checks are performed rather than every single possible array check.

3.2.6 Internal functions which use arrays

There are several functions internal to Modula-S which can also be used with open arrays. These functions usually simply include special case routines to handle open arrays, and are also very straightforward.

FIRST, LAST, and NUMBER. Three of the more important functions associated with open arrays are `FIRST`, `LAST`, and `NUMBER`. These do not present any difficulty in adapting to open arrays, mainly because the open array size is already included on the stack. The `FIRST` function is the simplest, simply having to remove the open array descriptor and push 0 (zero) onto the stack, since open arrays almost always have an index start at zero. (The case of an *empty* open array is not allowed in Modula-S.) The `LAST` and `NUMBER` functions aren't much more complex. Both must remove the array pointer from the descriptor with one simple instruction. At this point `NUMBER` is finished because the value on top of the stack is the array size in elements. However, `LAST` must then subtract 1 from this value to obtain the index of the last element. These additions are extremely simple to make and add very little new code to the compiler.

SUBARRAY. The `SUBARRAY` function is another instance in which open arrays can be used, but it is interesting to note that `SUBARRAY` itself creates open array types. The size of the resulting subarray is rarely predictable, simply because `SUBARRAY` can be called with variables for the length; therefore it is necessary for `SUBARRAY` to return an open array type, as well as the open array descriptor on the stack. This case is similar to the use of open arrays in procedure calls; since the array elements are already in existence somewhere, they do not need to be created or copied into a memory heap.

If the original array is a fixed-length array, `SUBARRAY` will perform the various addressing operations and then produce a new open array descriptor at runtime, while returning a new open array type at compile-time. However, if the original array is variable-length then `SUBARRAY` can simply use the descriptor already on the stack, adjusting the array pointer and size as necessary. The same open array type is returned at compile-time since the open array's size is not (and cannot) contained in the type descriptor. The element-type information *is* contained, however, and thus the same open array type *should* be returned.

WITH. The `WITH` statement is yet another instance in which open arrays can be used; however open arrays are currently not allowed in this implementation

of the WITH statement. The necessary changes should be quite simple to make, and no additions or changes need to be made to the stack-machine because the array elements already exist somewhere else and do not have to be allocated in a heap.

The only real difference between normal array variables and open array variables is that instead of only having a reference to the original variable, the entire open array descriptor would be copied, allowing the new WITH variable to be used as a completely normal open array variable. This would allow all other code to remain the same, making the change very simple.

3.2.7 Procedure declarations and calls

The final major area of implementing open arrays deals with the declaration of open array types and the actual passing of open arrays. Because of the limits that are placed on open arrays, they may only be declared in procedure parameter declarations. This is simple to control, however, and does not present a problem. The most important change in the compiler must be made in the code to handle procedure calls, for this is where the open array variables are created. As each parameter is parsed, the compiler normally simply compares types for compatibility. But with the addition of open array parameters, the compiler must also perform an additional check. If the procedure parameter is of an open array type and the actual value being passed is a fixed-length array, an open array descriptor must be created by pushing the array's size onto the stack as well. If the actual value is a variable- length array then no conversion is necessary. This additional check is simple enough to add, and doesn't present any conflicts with the rest of the compiler.

3.3 Runtime Error Checking

One of the more important features of any programming language in general, and Modula-3 in specific, is the ability to respond to certain situations by checking for and/or raising a run-time error. As part of extending Modula-S to emulate Modula-3 more closely, several changes were made to include run-time error checking for several situations.

3.3.1 Values that exceed their bounds

The most simple and basic form of error checking that needs to take place in any compiler is to check for values that exceed their limits. There are several situations in which this can arise, and most are handled in Modula-S. Because the original architecture of the stack machine had no methods for making runtime checks, several new opcodes were created to accomodate this function. The two opcodes **chkh A** and **chkl A** (see Appendix A for complete description) were the first opcodes to perform a simple runtime error check of the value on top

of the stack, and other opcodes were later created to handle other situations (discussed later).

The simple error checking opcodes `chkh` and `chkl` are extremely useful and are used throughout the compiler to check runtime values. These opcodes are used mainly for checking limits on subrange and enumeration types including the predefined `CHAR` and `CARDINAL` types, and array indexing.

3.3.2 Fixed-length array indexing

It is very important that array index values remain in their proper bounds to keep programs from addressing areas of memory they shouldn't. The addressing of arrays with index values is handled in Modula-S by the procedure *selector*, which loads the address of the array, adjusting the address according to the index values, if any. In the original Pascal-S compiler, this procedure contains no method for checking index values, but in Modula-S these checks are generated immediately after the index values are loaded on the stack. The method is quite straightforward, and is a simple yet effective way to maintain the integrity of the index values.

3.3.3 Subranges and enumeration types

A slightly more complex problem arose with the addition of subranges and enumeration types to the compiler. Variables of subrange or enumeration types have very strict bounds set upon them, and those bounds must be maintained by the runtime code.

The main concern with these variables arises when values are assigned to them, either from a constant expression or from another variable. In any case, the answer lies in determining the types of the expressions on either side of the assignment operator. The destination variable's bounds are the crucial ones, for the value being assigned must conform to those bounds. Therefore, all that needs to be done is to load the value of the right-hand-side, make the usual check for basic compatibility (done only by the compiler), and then generate code to check that the limits of the left-hand-side type are obeyed. The procedure *errchk* was created to perform this function. The assignment section of the compiler simply needs to pass the type of the value on top of the stack, and *errchk* then generates lower and upper limit checks. Other sections of the compiler also generate error checks if a value is absolutely known to be `CARDINAL`, etc., such as the `SUBARRAY` function handler, but the majority of the compiler uses the *errchk* procedure to generate error-checking code for this kind of situation.

3.3.4 Variable-length array checking

Including variable-length arrays in Modula-S created a whole new section of runtime error checking that must be performed by the generated code. There

are now several instances in Modula-S where the size of arrays is not a preset value, such as using the SUBARRAY function with variable lengths or through the passing of open array parameters in procedures. Because of this changing, unpredictable size, there needs to be some sort of runtime check to make sure arrays are of the same size. (The elementary type checks can be made at compile time because the array type can be determined at that point.) This form of error checking only takes place in two main situations: array comparisons, and array assignments.

Although code must be added for checking open array sizes, the check itself is quite simple because of the open array's structure on the stack. Because the open array's size in elements is already on the stack, the code simply needs to load the size of the other array on top of the stack and perform the error check.

Several additions were made to both the opcode set and the compiler code to fully implement this checking. First, the opcode `ckeq` was created to compare the top two values on the stack. Then a new procedure called *checkarrays* was added to generate any error checking code. The code can call *checkarrays*, passing the two array types as parameters. The procedure will then create code to check the array sizes if either or both of the arrays are open, and to convert the open array entries to normal array entries on the stack. By converting the open arrays to normal arrays, the compiler is kept much simpler because only a very few special cases need to be handled. The most important of these is if two open arrays are being used for a comparison or an assignment. In this case the size of the arrays in words is left on top of the stack as well to allow the following `eqal` or `move` instructions (for the comparison or assignment) to work with the correct number of words.

Checking open array index values also called for the addition of a new opcode, `ckls`. Because the open array size in elements is already loaded on the stack, it is very simple to check the index value against the array size once it is also loaded on the stack. The procedure *selector*, which handles most array indexing, generates the `ckls` opcode when needed.

3.3.5 Checking arguments passed to internal functions

Several other error checking routines must be included for specific internal Modula-S functions, such as INC, DEC, and most notably SUBARRAY. These functions must perform checks on their arguments to insure that their arguments are allowed with the types they work with, and that the program won't act in an unpredictable way. Functions such as INC, DEC, and READ are all very simple to check. The variable, once it is updated by the function, is simply reloaded if necessary (such as with the READ function) and checked to make sure it is still within the bounds of its type, once again using the procedure *errchk* to generate the error checking code. The most extensive error checking is performed by the SUBARRAY function, which must perform checks to insure that the subarray is indeed inside of the original array. This error checking is relatively

straightforward, as is all the error checks used with the internal functions.

3.3.6 Pointer values and reference types

Because the only pointers that are used right now are generated by the compiler itself, no methods for runtime error checking operations on pointers were researched. However, they would probably be very similar to the operations that are currently performed, such as checking if a pointer is `NIL` and/or what type the reference points to before dereferencing it.

3.4 Large-Variable Operations

Modula-3 contains many different features to use with the various data structures, and to make Modula-S more useful, many of these features are also included in the Modula-S compiler. Including the various large-variable constructors and operations leads to some interesting problems to be addressed, and many changes were made to the stack-machine as well as the compiler code itself to add the specified operations.

3.4.1 Original functioning of the Pascal-S compiler

The original Pascal-S compiler from which the Modula-S compiler was taken has very few supports for data structures. Basically, array and record types could be declared and assigned to each other, as well as assigning values to the various elements of the structures. Omitted were any method of comparing large variables, as well as any array or record constructor to assign values to the elements. Also, large variables were not allowed to be returned by procedures. In the process of adding these simple yet important functions, several new opcodes were created, and as the work progressed even more changes were made.

3.4.2 Array and record comparisons

The first area to be addressed was the comparison of larger variables. Many opcodes exist for the comparison of one-word variables: the simple `eqli` (=) and `neqi` ($\neq$), and also the `gtri` (>), `geqi` ($\geq$), `lssi` (<), and `leqi` ($\leq$) opcodes to perform any necessary check on two one-word values. However, no opcodes were originally present for comparisons of larger variables. Therefore two new opcodes were created, `eqla` (=) and `neqa` ($\neq$). To make a comparison of two multi-word variables, pointers to each variable would be pushed onto the stack. Then by using an `eqla A` or `neqa A`, with A being the number of words in the variables, programs could now compare larger size variables. This method worked quite well for the time it was used, however the introduction of more large-variable operations led to several necessary changes in these opcodes.

3.4.3 Array and record constructors

Once large-variable comparisons were completed, the next area to be addressed was that of array and record constructors. These presented a more difficult problem to solve, and many new additions and changes were made. The most complicated issue arose from the fact that ALL large-variable operations were done solely with pointers at this time, but if array and record constructors were to be introduced, the actual array or record would have to be stored and/or manipulated on top of the stack. In the evaluation of *arraytype*{`1, 2, 3, 4, 5`}, for instance, the actual array would have to be generated on top of the stack and then stored somewhere, without being able to use a pointer as before.

Several new opcodes were created at this point. Firstly, the `sta` opcode was created to store a preset number of words from the stack into a specified address. This was necessary so that once the actual array is generated on the stack it can be immediately stored somewhere. Secondly, the `rev` opcode was created. An interesting effect is that in the evaluation of an expression such as *arraytype*{`1, 2, 3, 4, 5`}, the expression must be parsed from left to right, leading to the actual values being stored in exactly the opposite order from what the need to be. Rather than creating extra code to manipulate the stack after the values are loaded, the `rev` opcode simply reverses the order of the top specified number of words.

Evaluating array constructors is quite straightforward; the most complicated aspect is taking precautions to make sure that elements made up of more than one word don't become reversed by the `rev` opcode at the end of the construction. Take the construct *arraytype*{*recordvar*, ..}, for instance. After *recordvar* has been loaded onto the stack, it would then be reversed so its words are backwards. Then, when the entire construct has been loaded onto the stack, the final `rev` instruction will reverse the words once again, putting them into the proper order.

Evaluation of record constructors is slightly more complicated, but only because of the way the fields are stored in the compiler itself. Record fields are stored in reverse order, so a recursive procedure was needed to find the end of the list of fields and step back towards the start as each specific field is evaluated. Once again, though, this is actually quite simple to understand. As is the case with procedure calls in Modula-S, record constructors cannot have the various field names specified; they must be in the proper order.

3.4.4 New large-variable comparisons

With the addition of array and record constructors, large variable comparisons had to change once again. Because the large variables being compared are no longer always pointers, such as `IF` *array* = *atype*{`TRUE, FALSE, TRUE`}, the original large variable comparison opcodes became obsolete. There are now four distinct possibilities in an array comparison:

1. Both are pointers to the arrays.
2. The first is only a pointer and the second array is actually on the stack.
3. The first array is actually on the stack and the second is only a pointer.
4. Both arrays are actually on the stack.

Four opcodes, `eqa1`, `eqa2`, `eqa3`, and `eqa4`, were created, each one being for one of these situations. Because the compiler must know if the large variable is actually on the stack or not, a new flag was added to the procedure which evaluates expressions. If the large-value is actually on the stack then the flag is `TRUE`, otherwise it is `FALSE`. In this way the compiler can tell exactly what situation it is facing and produce the corresponding opcodes. These opcodes, like the `sta` opcode, take one argument consisting of the number of words to compare. Because variable-length arrays were not present at this time, the number was easy to find and was included as an argument.

3.4.5 The introduction of open arrays

This system of comparisons, etc., for large variables works quite well for fixed-length variables, but yet more changes were made to create the flexiblity needed for variable-length variables, namely open arrays. Once open arrays were included in the Modula-S compiler, all large-variable opcodes were once again changed so the number of words to manipulate would be left on top of the stack rather than as an argument of the opcode. In this way the number of words moved/compared/stored can be changed at runtime, allowing the necessary flexiblity for including open arrays. This, however, led to another needed change in the Modula-S compiler. A very important aspect of the compiler is its ability to keep track of the stack pointer so that the various operations will know what part of memory to work with. However, leaving the number of words on the stack makes it so the compiler no longer can keep track of the stack. To account for this, a new procedure was added specifically to generate these opcodes that adjust the stack in an "unpredictable" way. At every point where one of these opcodes is generated, there is some specific type which the opcode is being used on. This type is passed to the procedure with the opcode to generate, and then the procedure can update the stack pointer in the normal way. This allows the opcodes to be used with variable-size structures while still being able to keep track of the stack pointer.

3.4.6 Procedures returning large variables

Originally the Pascal-S, and thus the Modula-S, compiler did not allow procedures to return values larger than one word. However, since this is an important and useful feature of Modula-3, several simple changes were made to allow this situation to take place. When a procedure which returns a large variable is

called, a space is made on the stack for the entire variable to be left there. Once the procedure returns the large variable, the value is immediately copied to this area. The compiler then sets the on-stack flag to TRUE because the actual structure is on the stack, and returns from the procedure as it would from any other procedure. The implementation is very straightforward, and it fits well with the other additions and changes which were introduced in this area.

3.5 IF-blocks

One of the more interesting situations encountered in the Modula-S compiler was that of implementing the ELSIF statement in IF...THEN blocks. The original Pascal-S compiler did not contain support for this statement as it is not part of Pascal, and therefore it had to be added to the Modula-S compiler. The implementation of the statement is not difficult to understand; the compiled code must check if the boolean expression is true, and if not then it must jump past the following statements. However, the difficulty arises in keeping track of the jumps after each block of statements to make sure they all jump to the proper location after the IF-block code. For example, in the block

IF $b1$ THEN $s1$ ELSIF $b2$ THEN $s2$ ELSE $s3$ END ,

the compiler must generate a jump to the end of the IF-block after every statement block except the last one ($s1$ and $s2$) so the code will not execute an inappropriate section of the block. The difficulty in this is that there can be any number of ELSIF branches in the block, each of which must be kept track of so the jumps can all be pointed to the correct location once the entire block is compiled.

There are many different ways to address this problem of keeping track of the different locations in the compiled code, some more efficient than others. Several of the more obvious solutions lie in the use and maintenance of a linked-list of code locations by the compiler; the compiler would store the location of each ending jump in the list and then point all jumps to the end of the block once it is fully compiled. However there is a much simpler solution, one that does not requre the use of any linked lists or other variables, but that uses the methods of Modula-3 to effectively and simply compile IF-blocks.

This method lies in the fact that any IF...THEN...ELSIF...END block can be represented by a number of nested IF...THEN...ELSE...END blocks. Take the following, for example:

IF...THEN...ELSIF form

```
IF b1 THEN
  s1
ELSIF b2 THEN
  s2
ELSIF b3 THEN
  s3
ELSE
  s4
END
```

IF...THEN...ELSE form

```
IF b1 THEN
  s1
ELSE
  IF b2 THEN
    s2
  ELSE
    IF b3 THEN
      s3
    ELSE
      s4
    END
  END
END
```

Although the code on the left is far more succinct than the code on the right, they are completely identical in their functioning. So instead of creating a more compilcated procedure in the compiler to create and maintain a linked-list of code locations, the compiler contains a very simple recursive procedure which handles the code on the left as if it were structured like the code on the right. This procedure only compiles the basic IF...THEN...ELSE block, and any time an ELSIF branch is discovered in an IF...THEN ...ELSE block, it is treated as if it is its own separate, nested IF... THEN...ELSE block. The procedure calls itself to handle the new block, and once the new block has been handled (and this new block can contain any number of "nested IF...THEN...ELSE blocks," depending on how many more ELSIFs there are in the original block), the compiler knows that it has handled all ELSIFs in the block and can then correctly generate the jump opcode.

To make the process more understandable, an example will be given. In the code seen earlier, the compiler will handle it thus:

What program contains

```
IF b1 THEN
  s1
ELSIF b2 THEN
  s2
ELSIF b3 THEN
  s3
ELSE
  s4
END
```

What compiler "sees"

```
IF b1 THEN
  s1
ELSE
  IF b2 THEN
    s2
  ELSE
    IF b3 THEN
      s3
    ELSE
      s4
    END
  END
END
```

What happens:

1. The start of the IF-block is encountered. The procedure is called for the first time.
2. The boolean conditional $b1$ is processed, and then the statement block $s1$ is also processed.
3. The first ELSIF is encountered. The compiler treats it as if it is an ELSE statement followed by a new IF-block and the procedure calls itself.
4. Under the new iteration of the procedure the next boolean conditional $b2$ is processed and the statement block $s2$ is generated.
5. The second ELSIF is encountered. The procedure, once again, treats it as if it is an ELSE statement followed by a new IF-block, and the procedure calls itself again.
6. Under the second iteration of the procedure the third boolean conditional $b3$ is processed and the statement block $s3$ is generated.
7. The ELSE statement is encountered, and is treated as if it is part of the "second nested IF-block."
8. The statement block $s4$ is generated as part of the "second nested IF-block."
9. At this point the second iteration of the procedure has completed one IF...THEN...ELSE block and updates the proper location for the jump to point to, making the end of $s3$ jump to the right spot. Notice that it will point to the very end of the original IF-block. After this is completed, the

second iteration of the same procedure returns to the first iteration which performs the same task, thus pointing $s2$ to the proper location. After the first iteration is complete, it returns to the original procedure, which then points $s1$ to the proper location. At this point the original procedure exits to the rest of the compiler, the entire IF-block being processed and generated correctly.

Although it is rather complex in its actual operation, the code required to implement IF-blocks in this way is almost half of what it would require to use a linked list instead.

3.5.1 Case blocks and this method

The CASE block presents a very similar problem to the implementation of the ELSIF statement in IF-blocks. There is probably a very similar way in which a CASE block could be implemented so that a very small, recursive procedure would be called every time a new branch in the CASE block is encountered. However, instead of creating a similar method for case blocks, the original idea of keeping a linked-list of code labels to update *after* the entire block is processed was used instead. The more elegant solution would probably be the one using a recursive procedure, for the introduction of a linked list into a procedure creates other problems. Using recursive procedures is very nice because the variables in each iteration of the procedure are separate from each other and no precautions are needed to insure that none are modified incorrectly. However, when a linked list is maintained and the procedure does not call itself, one must be very careful that all variables are used correctly to keep from modifying one which will result in incorrect compilation. Often this consists of introducing several other variables to keep track of other information, leading to the further complication of the procedure.

3.6 The WITH statement

Yet another statement not supported by the original Pascal-S compiler was the WITH statement, which can be used to bind new names to variables or values. The statement itself is quite easy to understand, but introducing it into the Modula-S compiler required several extensive changes to the expression-evaluating procedures.

3.6.1 Writable designators

The problem inherent in the implementation of the WITH statement is that each expression being assigned a name can be handled in two different ways: as an old variable being assigned a new name, or as a constant with a runtime-specified value. The difficulty lies in determining which of these cases is the proper one, and this requires knowing whether the expression being assigned is a writable

designator or not. If the expression is not a writable designator then it cannot be modified in the scope of the WITH block and therefore should be handled as the latter case. However, if it *is* a writable designator then it *can* be modified and therefore should be handled as the former case. So the solution of the problem lies in creating a writable-designator flag which the expression procedure can modify, depending on whether the previously-evaluated expression is a writable designator or not.

3.6.2 Changes made in the expression-handlers

The changes made in the tree of expression-evaluating procedures are quite extensive, but they are also straightforward. The original procedure for evaluating an entire expression must return a flag (called *wd* in the Modula-S compiler) telling whether the expression is a writable designator or not, but the determination cannot be made simply at that level. The underlying procedures which evaluate the terms of the expression and the factors of each term must be careful to keep *wd* updated at all times. If any factor of an expression is not a writable designator then the entire expression is not a writable designator, and if there is more than one term then the expression is also not writable. Therefore the *factor* and *term* "subprocedures" must also be able to pass the flag back and forth to insure that the *expression* procedure returns the correct value in the flag. Once again, the changes needed to include a new flag are quite extensive, but also are very simple to understand.

Another major change was made in the generation of the expression by the *expression* procedure to accomodate the new writable-designator flag more fully. Often when the compiler is interested in whether an expression is a writable designator or not, it is not always necessary to load the actual value of the expression onto the stack, only the pointer to the value, and sometimes the latter is even preferable. Therefore the tree of expression-handling procedures was modified once more to load only the address of an expression factor if it is a writable designator. If the expression is discovered to consist of more than one factor or some other situation arises then the writable-designator flag can be changed to FALSE and the value of the factor loaded instead of just the address. Also, from that point on the *factor* procedure must check after every factor is evaluated so that if the writable-designator flag is already FALSE at that point, the factor itself is loaded onto the stack and not just its address. In the case when the compiler needs only the value of the expression and not the writable-designator information, the compiler can still call the *expression* procedure. Then if the expression is a writable designator the compiler can simply load the value from the address loaded on the stack; in the other case the value is already there so no extra effort is needed.

3.6.3 These changes and the WITH statement

After these changes were added, it became very easy to implement the WITH statement. The compiler can load each expression, determine if it is a writable designator or not, and then treat each expression accordingly. New identifiers are created by the compiler for each expression processed, and the proper names, addresses, and attributes are given to these new variables. Upon completion of the WITH statement these new variables are removed from the stack, and the program is able to continue as if the identifiers never existed. The changes in the expression procedures are also useful in many other situations as well, even in simple assignment statements, because the new writable designator flag presents a much simpler way to determine if an expression can indeed be written to.

4 Conclusion

After creating and expanding the Modula-S compiler, several interesting facts and methods have become clear to me. Although the scope of the compiler is extremely limited compared to normal Modula-3 compilers, it is still a useful and effective compiler, creating partially optimized code after only one pass over the program code. The stack-machine has also changed and grown over the project, yet it is still very simple and basic in most regards.

There are several things which could be improved or changed in this Modula-S compiler in the future. In most cases, the compiler is rather efficient in its performance, although there are, no doubt, several sections of the code which could be streamlined or otherwise improved. Also, the peephole optimizer which is used is also not the most efficient possible, leaving out many optimizations that could be made to greatly reduce program code. However, this could be an entirely separate area of research in itself, so the problem was not closely scrutinized. Of course, both the compiler and the Modula-S programming language can both be expanded to more closely mimic Modula-3, starting most likely with the inclusion of pointer types. Many more changes and additions could be made to the compiler as well, such as the addition of objects and low-level programming functions. As it is, though, the Modula-S compiler is still useful for smaller programs, and it is very useful for studying various methods of implementing the different statements and commands in Modula-3.

A The Stack-Machine

The code generated by the compiler is intended for an imaginary stack machine. The machine is very simple, consisting of a linear block of memory, two registers, and a flag. The registers consist of the stack pointer which points to the top of the stack, and the instruction pointer which points at the next instruction

to execute in the code. The stack grows from the top of memory downward. The instruction pointer starts at address 0 and is usually incremented after each instruction is executed, except for several opcodes which directly affect the instruction pointer. There is also one flag which is used to halt the program's execution upon completion or a runtime error. Because the machine is imaginary, it is extremely flexible in both the opcodes it allows and the operation of the machine. An interpreter is also present in the compiler to run the generated code, and can be adjusted at will for different size programs, new or more complete opcodes, or any variety of other additions.

The stack machine currently does not have any support for a heap, but it would be necessary to have one if dynamic variables were to be included in the Modula-S programming language. This was to be a specific area of research, but unfortunately there wasn't the time for it. Much of the heap maintenance, such as garbage collection, etc., would take place in the interpreter code, so there would probably not be many more opcodes added to those currently supported.

There are currently 55 opcodes which can be grouped into various functions of arithmetic operations, error checking, memory loading and storing, boolean operations, comparisons, stack operations, program control, I/O, and a few miscellanious other functions. A grouped list of currently supported opcodes is listed below. (Because I was intent on limiting the set of opcodes to the fewest possible, I have also included "substitutions": sections of code that perform the identical function but with simpler opcodes. However, I left the more complex opcodes in to maintain speed and simplicity.)

SP = stack pointer to top of stack (stack is at SP, SP+1, ...)
IP = instruction pointer (points to the next instruction to be executed)
(NS) = Not supported at this time

OPCODE FUNCTION

Arithmetic operations

`add` Pop two integers from the stack. Push their sum onto the stack.
SP := SP + 1

`addc A` Add constant A to the integer that is on top of the stack.
SP remains unchanged

Same as:
```
ldc A
add
```

`div2` Pop an integer from the stack. Push *integer value* DIV 2 onto the stack (basically shift the integer one bit to the right).
SP remains unchanged

Same as:
```
ldc 2
divd
```

`divd` Pop two integers from the stack. Push their quotient onto the stack (value = *second integer popped* DIV *first integer popped*).

SP := SP + 1

`mul` Pop two integers from the stack. Push their product onto the stack.

SP := SP + 1

`mulc A` Multiply the integer that is on top of the stack by constant A.

SP remains unchanged

Same as:
```
ldc A
mul
```

`neg` Reverse the sign of the value on top of the stack.

SP remains unchanged

Same as:
```
ldc -1
mul
```

`rem2` Pop an integer from the stack. Push *integer value* MOD 2 onto the stack (basically ANDs the integer with 1).

SP remains unchanged

Same as:
```
ldc 2
remd
```

`remd` Pop two integers from the stack. Push their modulus remainder onto the stack (value = *second integer popped* MOD *first integer popped*).

SP := SP + 1

Boolean operations

`andb` Boolean AND. Pop two booleans from the stack. Push *bool1* AND *bool2* onto the stack.

SP := SP + 1

`notb` Boolean NOT. Pop a boolean from the stack. Push NOT *bool* onto the stack.

SP remains unchanged

Same as:
```
neg
ldc 1
add
```

`orb` Boolean OR. Pop two booleans from the stack. Push *bool1* OR *bool2* onto the stack.

SP := SP + 1

Comparisons

`eqa1` Pop *array size* from stack. Pop two addresses from stack. Push 1 if the arrays of *array size* words are equal (exactly alike), push 0 otherwise.

SP := SP + 3

`eqa2` Pop *array size* from stack. Pop an address from stack. Pop an array of *array size* words from stack. Push 1 if arrays are equal, push 0 otherwise.

SP := SP + *array size* + 1

`eqa3` Pop *array size* from stack. Pop an array of *array size* words from stack. Pop an address from stack. Push 1 if arrays are equal, push 0 otherwise.

SP := SP + *array size* + 1

`eqa4` Pop *array size* from stack. Pop two arrays of *array size* words from stack. Push 1 if arrays are equal, push 0 otherwise.

SP := SP + 2 * *array size*

`eqli` Pop two integers from the stack. Push 1 if they are equal or 0 if they aren't equal.

SP := SP + 1

`geqi` Pop two integers from the stack. If *second* $\geq$ *first* then push 1, else push 0.

SP := SP + 1

`gtri` Pop two integers from the stack. If *second* > *first* then push 1, else push 0.

SP := SP + 1

`leqi` Pop two integers from the stack. If *second* $\leq$ *first* then push 1, else push 0.

SP := SP + 1

`lssi` Pop two integers from the stack. If *second* < *first* then push 1, else push 0.

SP := SP + 1

`neqi` Pop two integers from the stack. Push 1 if they are not equal or 0 if they are equal.

SP := SP + 1

Error checking

`chkh A` Check if the value on top of the stack is higher than A. If so then make a runtime error.

SP remains unchanged

`chkl A` Check if the value on top of the stack is lower than A. If so then make a runtime error.

SP remains unchanged

`ckeq` Check if the two values on top of the stack are equal. If not then make a runtime error.

SP remains unchanged

`ckls` Check if the value at m[SP+1] is lower than m[SP]. If not then make a runtime error. Pops top value off of stack.

SP := SP + 1

Memory loading and storing

`copy` Pop *number of words* from stack. Pop an address from the stack. Copy the series of *number of words* words at that address onto the stack.

SP := SP - (*number of words* - 2)

`ldc A` Push constant A onto the stack.

SP := SP - 1

`ldg A` Load global value. Push the word at address A onto the stack.

SP := SP - 1

Same as:
```
ldc A
load
```

`ldl A` Load local value. Push the word at address SP + A onto the stack.

SP := SP - 1

Same as:
```
ldla A
load
```

`ldla A` Load local address. Push address SP + A onto the stack.

SP := SP - 1

`ldls` **(NS)** Load local value with value on top of stack. Replace m[SP] with SP + m[SP].

> **Note:** This opcode has not become necessary yet, and may never be needed at all. I thought I'd need it but then I found out that I didn't need it.

SP remains unchanged

`load` Pop address from the stack. Push the value at that address onto the stack.

SP remains unchanged

`move` Pop *number of words*. Pop two addresses from the stack. Move (duplicate) *number of words* words from the second address to the first address. (Push *from* address first, then push *to* address.)

SP := SP + 3

`sta` Pop *number of words* from the stack. Pop address from the stack. Store *number of words* words from the stack into that address.

SP := SP + *number of words* + 2

`stg A` Store global value. Store the word popped from the stack at address A.

SP := SP + 1

`stl A` Store local value. Store the word popped from the stack at address SP + A.

SP := SP + 1

`stor` Pop address from the stack. Pop the next value from the stack into the location at that address.

SP := SP + 2

Program control

`call A` Push IP onto the stack. Jump to address A.

SP := SP - 1

`exit A` Jump to the address that is on top of the stack. Pop A elements from the stack.

SP := SP + A

`halt` Terminate execution of the program.

SP's value no longer important

`jump A` Unconditional jump to address A (IP := A).

SP remains unchanged

`jumpz A` Pop one word from the stack. If it is zero then jump to address A.

SP := SP + 1

Stack operations

`adjs A` Increase SP by A.

SP := SP + A

`del A` **(NS)** Delete the word at SP + A from inside the stack. Move every value pushed on the stack up one address.

> **Note:** This is a "messy" opcode; an opcode like this wouldn't normally be supported, or even considered, in a real stack machine.

SP := SP + 1

`dupl A` Push the value on top of the stack A times, duplicating it A times.

SP := SP - A

`popa` **(NS)** Pop an integer from the stack. Pop that many words from the stack.

> **Note:** This opcode isn't really necessary, and it is also a very touchy opcode in that it can change SP in an unpredictable way. This is bad for the compiler code, which needs to know what SP is at all times.

SP is unpredictably changed

`rev A` Reverse the top A words on the stack.

SP remains unchanged

`sets A` Set SP to A.

SP := A

`swap` Swap the first 2 values on the stack.

SP remains unchanged

Input/output

`eol` **(NS)** Push 1 if currently at end of input line, push 0 otherwise.

SP := SP - 1

`rdc` Read one character from the keyboard. Store the character value at the address popped from the stack.

SP := SP + 1

`rdi` Read one integer from the keyboard. Store the integer at the address popped from the stack.

SP := SP + 1

`rdl` **(NS)** Reads a line of text. Read characters, pushing them onto the stack as they are received, until a RETURN character is encountered. Push the length of the string onto the stack.

> **Note:** This will be difficult because, once again, the stack will be affected in a way that is unknown. So perhaps it would be better if the opcode takes a pointer off of the stack and stores the string, as well as its length, at that pointer. Of course, that won't happen until the stack machine has a heap ...

SP is unpredictably changed

`wrc` Pop character value from the stack. Write the character to the screen.

> **Note:** With the addition of the `wrt` opcode, the `wrc` opcode is now kind of obsolete, but I discovered that using the `wrc` opcode will actually create *less* code if the string (or character) being written is less than 4 characters long. But it's kind of not worth the effort for the one or two instructions it will save ...

SP := SP + 1

`wri` Pop an integer from the stack. Write it to the screen.

SP := SP + 1

`wrl` Send a newline to the screen.

SP remains unchanged

Same as: `ldc` *value of newline character*
`wrc`

`wrt` Pop size of string from the stack. Pop an address from the stack. Write the string of characters, starting at the address, to the screen.

SP := SP + 2

Miscellaneous instructions

`for1 A` For the `FOR` statement.

SP remains unchanged

For the statement `FOR` i = a `TO` b `BY` n, does the following:

Is $n > 0$?
  If so then:
    Is $i > b$?
      If so then jump to A.
      If not then go to next opcode.
  If not then:
    Is $i < b$?
      If so then jump to A.
      If not then go to next opcode.

Same as:

```
        dupl
        ldc 0
        gtri
        jump L1
        ldl 2
        ldl 2
        leqi
        jumpz A
        jump L2
L1:     ldl 2
        ldl 2
        geqi
        jumpz A
L2:
```

`for2 A` For the `FOR` statement.

SP remains unchanged

For the statement `FOR` i = a `TO` b `BY` n, does the following:

Increment i by n. *Remember: n can be negative!*
Jump to A.

Same as:
```
dupl
ldl 3
add
stl 3
```

References

[vdS93] J.L.A. van de Snepscheut. *What Computing Is All About.* Springer-Verlag, 1993.